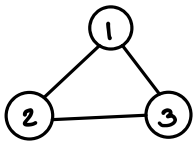
⇒ Introduction to graphs
⇒ Storing graphs
⇒ Graph searching: DFS & BFS.

Graph

- Defined by a set of vertices V and a set of edges E.
- Each edge consists of a pair of vertices.
- Vertices are also called nodes.
- We write $G=(V,E)$ for a graph with vertex set V and edge set E.

Examples:

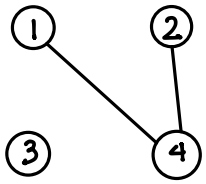
(a)



$V=\{1,2,3\}$

$E=\{\{1,2\},\{2,3\},\{3,1\}\}$

(b)



$V=\{1,2,3,4\}$

$E=\{\{1,4\},\{2,4\}\}$

Why study graphs and graphs algorithms? Graphs are used to model a variety of things like networks, maps, relationships, constraints ....

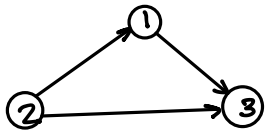
Undirected vs Directed graphs

⇒ Often, graphs are used to model relationships which are not symmetric (for example, one way roads in a map)

Idea Required edges to be directed (ordered pair of vertices)

Notation $\{x,y\}$ for undirected edge.
(x,y) for edge from x to y.

Example:



$V=\{1,2,3\}$

$E=\{(2,1),(1,3),(2,3)\}$

How are graphs stored in a computer?

⇒ Two standard (simple) data structures:

- Adjacency matrix
- Adjacency list

Adjacency matrix

- $G = (V, E)$ graph
- $n = |V|$ (the number of vertices in the graph).
- $V = \{v_1, v_2, \ldots, v_n\}$
- We create an $n \times n$ matrix A where:

$$A_{ij} = \begin{cases} 1 & \text{if there is an edge from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

- A is called the adjacency matrix
- For undirected graph, A is symmetric ($A_{ij} = A_{ji}$)
- Diagonal entries are often set to 0, since we typically work with simple graphs (no self-loops or multiple edges between vertices)

Adjacency List

- $n = |V|$ linked lists, one for each node.
- The linked for vertex v contains all the neighbors of v, that is all vertices u such that $\{u, v\}$ is an edge.

Comparison between Adjacency Matrix & Adjacency List

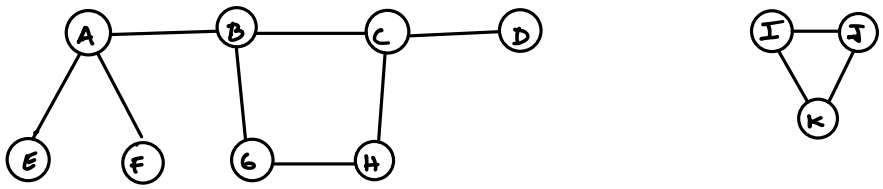
	Adjacency Matrix	Adjacency List
Memory	$O(\|V\|^2)$	$O(\|E\|)$
Edge Query	$O(1)$	$O(\|V\|)$
Find all neighbors	$O(\|V\|)$	$O(\|V\|)$

Exploring graphs: the undirected case

Let $G = (V, E)$ be an undirected graph

Is vertex $v \in V$ connected to vertex $w \in V$?

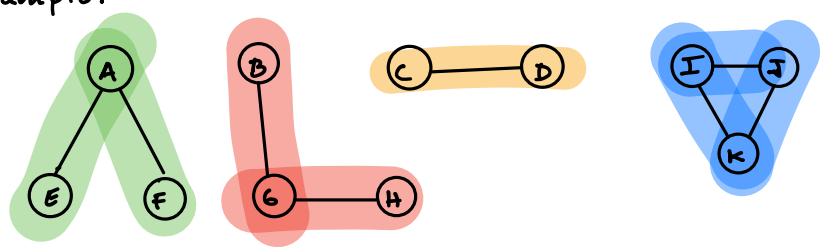
Example:



- A is connected to G
- A is connected to E
- D is not connected to K, I or J.

Definition A **connected component** is a maximal set of connected vertices.

Example:



There 4 connected components.

⇒ There two standard algorithms for finding the connected components of a graph:

- Depth first Search (DFS)
- Breadth first Search (BFS)

⇒ Once we know the connected components we also know whether v is connected w for any pair of vertices v and w.

⇒ DFS and BFS are very powerful algorithms and we will spend a few lectures learning about them an their applications

The Depth First Search algorithm

Intuition Can you explore a maze with a chalk and a string?

- Mark every intersection with the chalk to avoid revisits and the string to backtrack

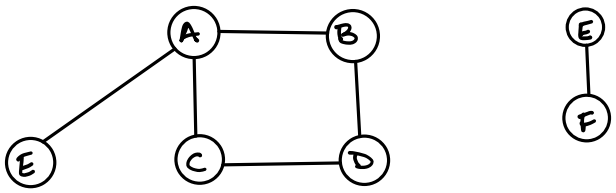
DFS follows the same idea:

- The graph is the maze, with vertices corresponding to intersections
- Boolean array (one boolean variable per vertex) – chalk!
- String can be modeled by a **stack**, to backtrack we pop from the stack.

The Explore procedure

- Used to identify **one** connected component.
- We assume the graph is given as an Adjacency List.

Example:



⇒ Explore from A : A → B → D → C (backtrack to A) → E

Pseudocode:

```
int[] visited = new int[|V|]
```

Explore

Input
- Graph $G=(V,E)$
- Vertex w
- Color to use as marker for connected component

Output Will assign color to all entries of visited reachable from w.

```
Explore ( G, vertex w, int color )
    Visited [w] = color
    for each edge {w,v} ∈ E
        if Visited [v] == 0
            Explore ( G, v, color)
```

Is Explore Correct?

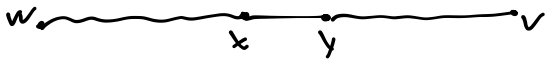
⇒ It can only reach vertices in the same connected component as w.

Lemma. Explore (G, w, color) marks all vertices in the connected component containing w with "color" in the visited array.

Proof. (By contradiction).

Suppose $\exists\, v \in V$ in the same connected component of w that is not marked by Explore.

$\exists$ A path P between v and w.



Let x be the last vertex in P marked with color.

Let y be the vertex after x in P.

Then {x, y} is an edge and must have been explored in the call Explore (G, x, color) call, which leads to a contradiction.

■

⇒ The Explore procedure reveals one connected component.

⇒ DFS works by repeatdly calling Explore.

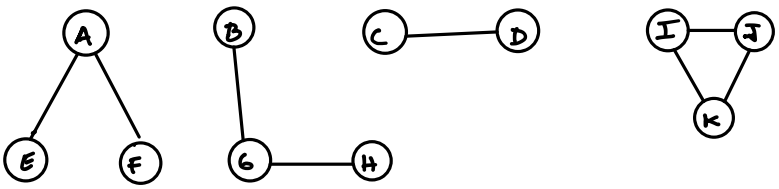
DFS

Input: A graph $G=(V,E)$

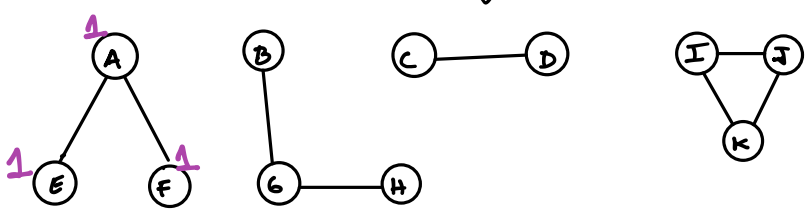
Output: An array containing the connected component information

```
int [] visited = new int [|V|]

int color = 1

for each w ∈ V
    if visited[w] == 0
        Explore (G, w, color)
        color = color + 1.

return visited.
```

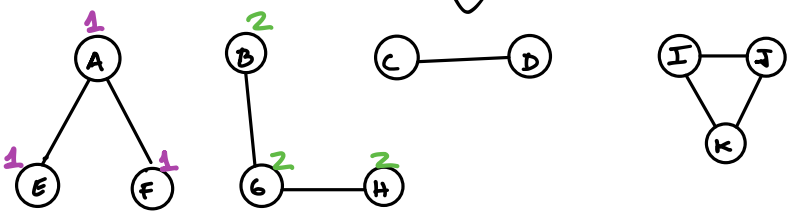
Example:

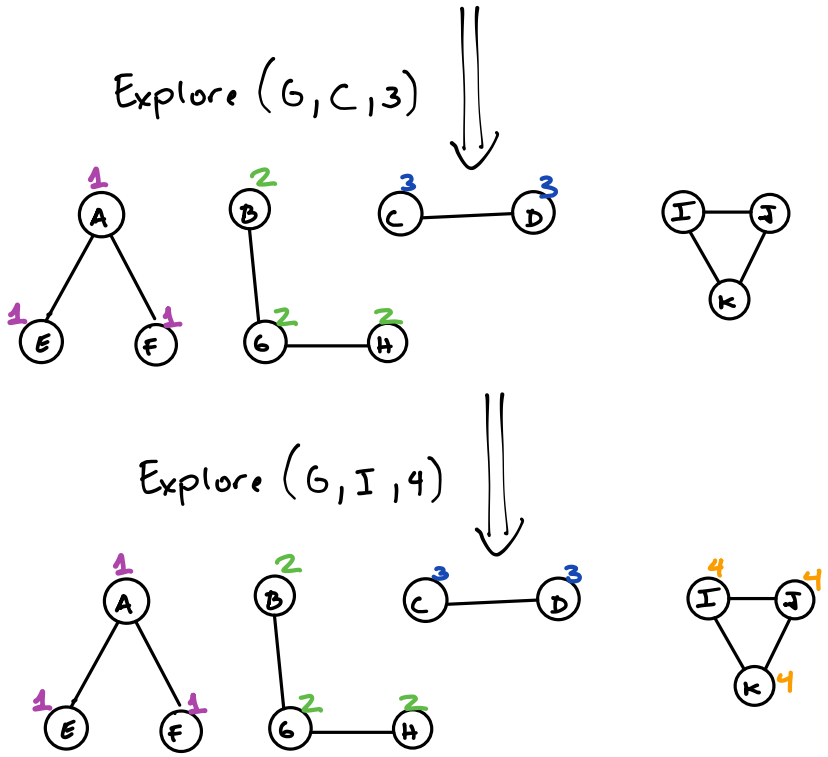


Explore (G, A, 1)



Explore (G, B, 2)





Running Time of DFS

⇒ Depends on whether the graph is given in Adjacency List or Adjacency matrix.

Adjacency Matrix

⇒ From every vertex, it takes $O(|V|)$ to check its neighborhood
⇒ Every vertex will be visited, so we get:

$$O(|V|^2).$$

Adjacency List

The for loop will take different times on different vertices, so we think globally:

⇒ Every entry of visited is marked once.

⇒ Every edge is considered at most twice, so:

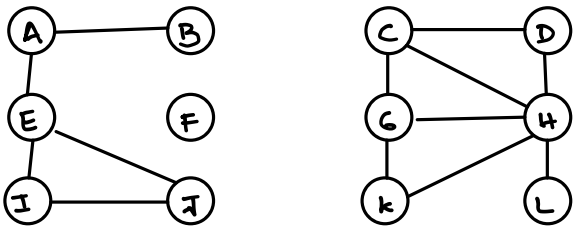
$$O(|V| + |E|).$$

⇒ Recall that $|E| \leq \binom{|V|}{2} \sim |V|^2$

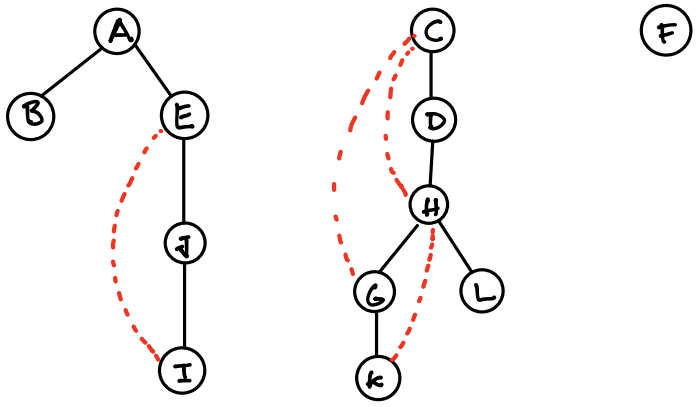
⇒ $O(|V| + |E|)$ is best possible, since we need to read the graph.

Edges types in DFS (undirected)

Example:



Suppose nodes are visited in lexicographical order.



⇒ Black (solid) edges are called **tree edges**

⇒ Red (dashed) edges are called **back edges**

⇒ Back edges correspond to **cycles**: a circular path from a vertex to itself.

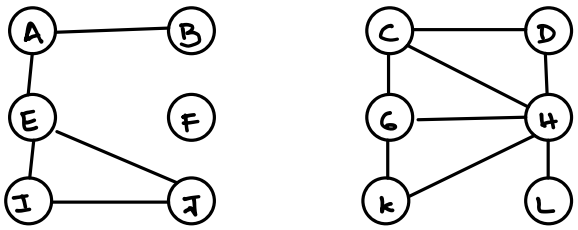
⇒ To find cycles, it suffices to find back edges which can be found with a simple modification to the explore procedure.

```
Explore ( G, vertex w, int color, vertex previous)
    Visited [w] = color
    for each edge {w,v} ∈ E
        if visited [v] != 0 and v ≠ previous
            Output {w,v} is a back edge or "cycle found"
        if visited [v] == 0
            Explore ( G, v, color, w)
```

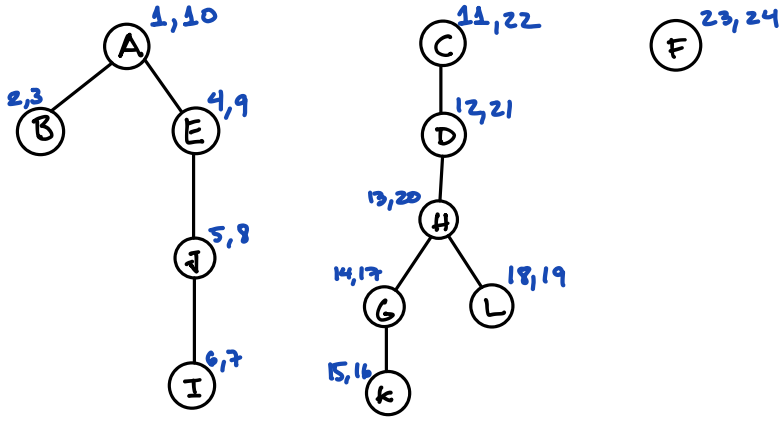
Pre and Post numbers (undirected)

Keep global counter (or clock) that is increased every time a node is visited for the first time and every time we leave a node for good.

Example:



Suppose nodes are visited in lexicographical order.



⇒ We shall see that these numbers have many applications.

⇒ How do we keep track of them? (again, simple modification to explore)

⊛ Before calling Explore, we initialize two arrays and maintain a global counter.

```
pre  = new int [|V|]
post = new int [|V|]
cnt  = 1
```

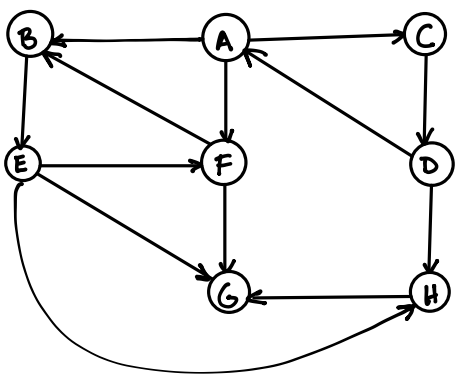
```
Explore ( G, vertex w, int color)
    visited[w] = color
    pre[w] = cnt
    cnt = cnt + 1
    for each edge {w, v} ∈ E
        if visited[v] == 0
            Explore (G, v, color)
    post[w] = cnt
    cnt = cnt + 1
```

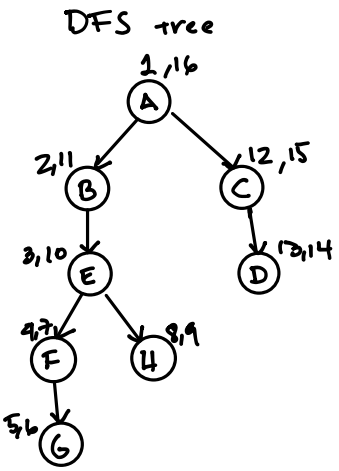
DFS in directed graphs

⇒ Exactly the same DFS algorithm works for directed graphs, we just need to consider edge directions.

⇒ Note that Explore (G, w, color) now marks all vertices reachable from w.

Example:





⇒ The edges used by the DFS tree are called **tree edges**.

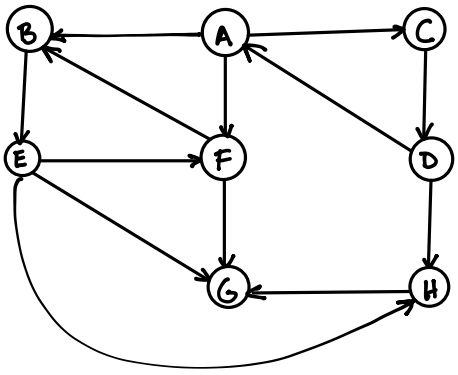
⇒ There are three other types of edges.

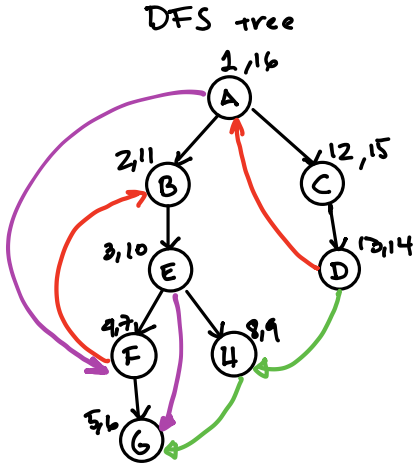
Back edges Lead to an ancestor in the DFS tree

Forward edges Lead to a non-child descendant.

Cross edges Lead to neither ancestor or descendant. (They have to lead to some node that has been completely explored; i.e., already has a post number).

Example:





- Back edges in red
- Forward edges in purple
- Cross edges in green

Pre visit, Post visit numbers and types of edges

Fact If vertex w is an ancestor of vertex v in DFS tree, then:

$$pre[w] < pre[v] < post[v] < post[w]$$

Consider the edge $w \to v$ (w,v)

- If $pre[w] < pre[v] < post[v] < post[w]$

  Then (w,v) is tree or forward edge.

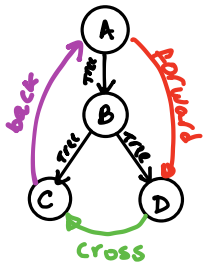
- If $pre[v] < pre[w] < post[w] < post[v]$

  Then (w,v) is a back edge

- If $pre[v] < post[v] < pre[w] < post[w]$

  Then (w,v) is a cross edge.

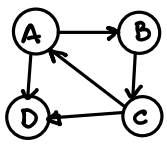
Summary



First Application of DFS: cycle detection.

Definition A cycle is a circular path $v_0 \to v_1 \to v_2 \to \dots \to v_k \to v_0$ with $v_i \neq v_j$

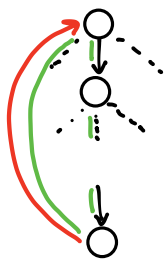
Example:



$A \to B \to C \to A$.

Lemma A directed graph has a cycle if and only if its DFS tree has a back edge.

Proof A back edge leads to an ancestor in DFS tree.:



The tree edges and the back edge form a cycle.

For the other direction, let $v_0 \to v_1 \to \dots v_i \to \dots \to v_k \to v_0$ be a cycle.

Let v_i be the first element of the cycle discovered by the DFS

All nodes from the cycle will be discovered by Explore(v_i) including v_{i-1}, and so (v_{i-1}, v_i) would be a back edge.

∎

Algorithm for detecting cycles:

1) Run DFS and assign pre and post numbers

2) Iterate through all edges (w, v) and check if:

   ⊛ $pre[v] < pre[w] < post[w] < post[v]$

3) If ⊛ holds for some edge, output cycle found; otherwise output "no cycle"

Second Application of DFS: topological sort

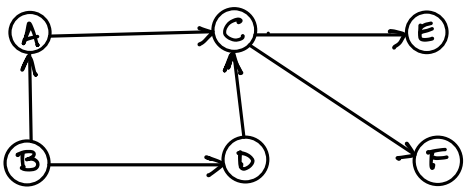
⇒ A directed graph without cycles is called a **DAG** (directed acyclic graph)

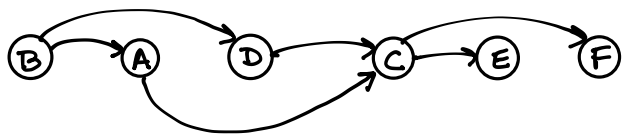
⇒ DAG's are useful to model hierarchies, time dependencies, etc.

Example: class prerequisites.

⇒ In DAG's, we would like to find orderings of the vertices such that all edges go from an earlier vertex to a later vertex

Example:





(All edges point left to right)

⇒ This type of ordering is called a **topological order** or a **linearization** of a DAG.

Fact. All DAG's have at least one topological order.

Proof. (By algorithm; we provide an algorithm to find one).

Fact In a DAG, every edge leads to a vertex with a smaller post number.

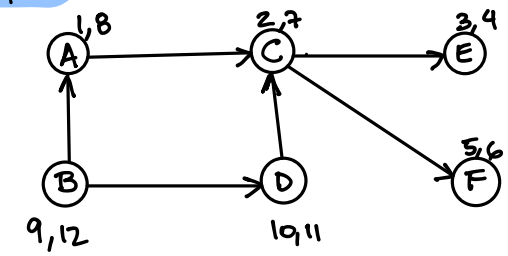
Proof. DAG do not have back edges, and all other types of edges satisfy this property. ∎

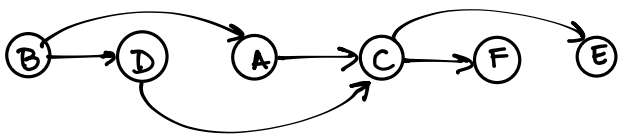
Algorithm for finding topological order.

1) Run DFS and assign pre and post numbers
2) Sort vertices in decreasing order of post numbers.

Running time : $O(|V| \cdot \log|V| + |E|)$

Example





⇒ We can improve running time with a simple modification to the **Explore** procedure

Idea: Sort at the same time we assign post numbers

```
int[] top_order = new int[|V|]    int index = 0;

Explore( G, vertex w, int color)
    Visited[w] = color;
    pre[w] = cnt;                 cnt = cnt + 1;
    for each edge (w,v) ∈ E
        if visited[v] == 0
            Explore(G, v, color)
    post[w] = cnt;                cnt = cnt + 1;
    top_order[index] = w;         index = index + 1.
```

Running Time of modified Algorithm: $O(|V|+|E|)$

Third application of DFS: finding strongly connected components

Connectivity in directed graphs

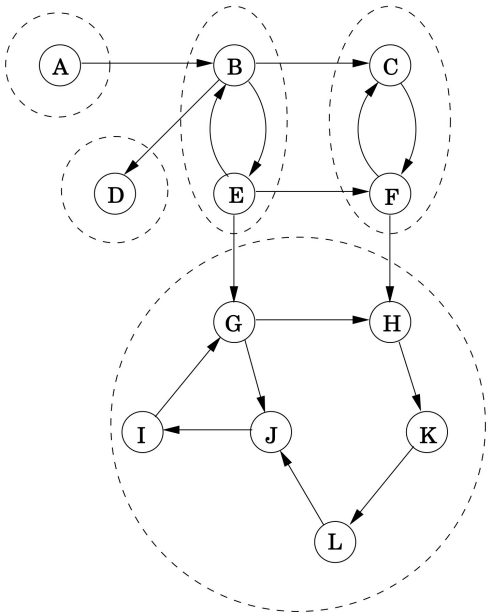
There is a path from A to C, but not the other way around.

Definition In a directed graph $G=(V,E)$, we say that vertices w and v are **connected** if $\exists$ path from w to v **and** from v to w.

⇒ This relationship ("connected to") defines an equivalence relation that partitions the vertex set of the graph G.

⇒ Each element from the partition is called a **strongly connected component**.

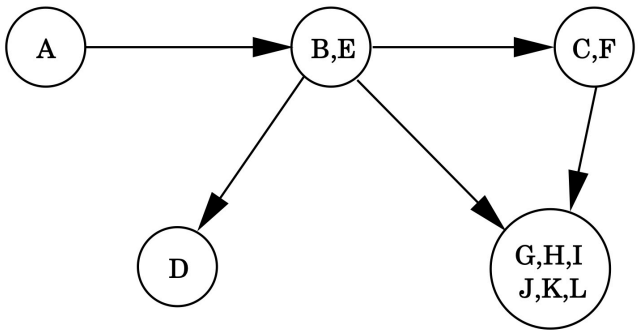
Example:



→ There are 5 strongly connected components

Goal Find all strongly connected components (efficiently).

Consider the following graph (often called a "meta graph", where vertices correspond to strongly connected components.



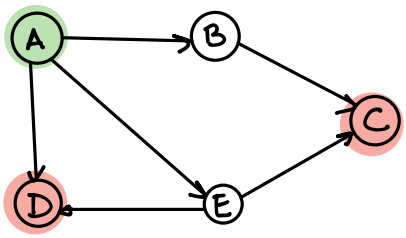
(*) Each node of the meta graph is a strongly connected component.

Fact The meta graph is a DAG.

Proof. If there is a cycle in the metagraph, then the strongly connected component in a cycle would be merged together. ∎

Definition In a directed graph, a **sink** is a vertex with no outgoing edges and a **source** is a vertex with no incoming edges.

Example:



- A is a source vertex
- C and D are sinks

Idea for algorithm If Explore is started from a vertex that is in a strongly connected that is a sink in the metagraph, then it will discovered exactly one strongly connected component.

The Algorithm will repeat the following steps

1) Find a vertex in sink strongly connected component.
2) Run Explore from this vertex
3) Remove strongly connected component found and repeat.

Key challenge: Find vertex in sink strongly connected component.

⇒ There is no easy efficient way of finding a vertex in a sink SCC. (Henceforth, I use SCC for strongly connected component).

⇒ However, finding vertex in source SCC is easy using the following fact.

Fact Suppose A and B are SCC's and that there is an edge from a vertex in A to a vertex in B. Then, the vertex with the largest post number is in A.

Proof Two cases. Suppose a vertex v from A is visited first. Then Explore would discover all vertices from B before finishing and assigning a post number to v.

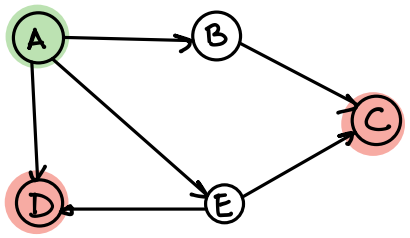
In the second case, suppose a vertex $w \in B$ is the first to be visited from $A \cup B$. Then Explore will discover all B, but nothing from A. So, all vertices from A have a larger post number. ∎

⊛ A **key consequence** of this fact is that the **vertex** with the **largest post number** must be in a **source SCC.**

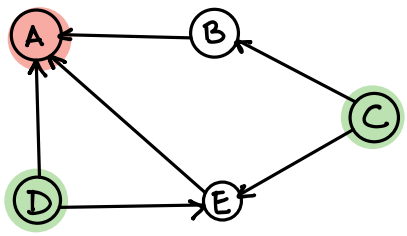
How to use this? **Idea:** Consider the reverse graph G^R.

⇒ G^R has the same vertex set as G, but the direction of every edge is reversed.

Example:

$G =$ 

Sources: A
Sinks: D, C

$G^R =$ 

Sources: D, C
Sinks: A

Facts about G^R:

1) It has the same SCCs as G.

2) In the meta graph of SCC of G^R, every source vertex (or SCC) corresponds to a sink vertex in G.

⇒ Therefore, if we run DFS on G^R and choose node with highest post number, it will be in a sink SCC for G!

⇒ So, we can find one vertex in a sink SCC of G. What next?

⇒ Remove sink SCC, and choose the unexplored node with the largest post number.

⇒ This node belongs to a sink SCC in the remaining graph by Fact 1 (above).

Final Algorithm

Input: Graph $G=(V,E)$ in adjacency list.

(1) Build adjacency list for G^R

(2) Run DFS in G^R, assign pre/post numbers and output nodes sorted from largest to smallest post number. Let

$$v_1, v_2, \ldots, v_n$$

be the ordering of the vertices.

(3) Run DFS on G using this ordering. That is,

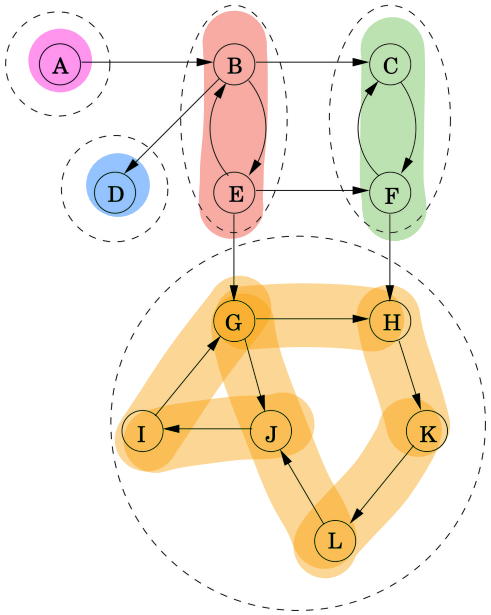
```
visited = new int [n]
color = 1

for i=1 to n
    if visited [i] == 0
        Explore (G, vi, color)
        color = color+1
```

Running time: Each step above, (1), (2), (3) and 4, takes $O(|V|+|E|)$, so overall, we get optimal $O(|V|+|E|)$.

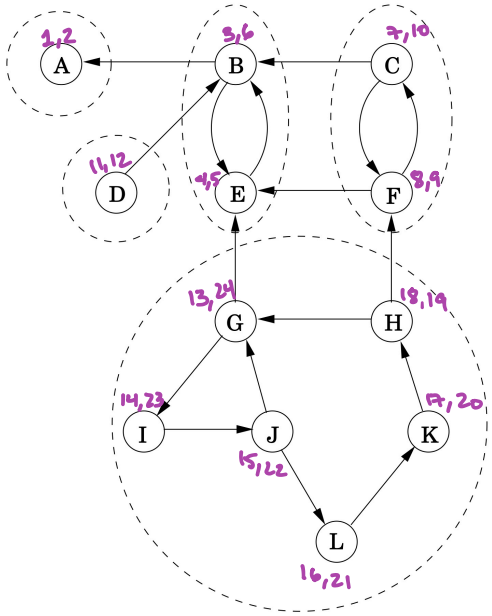
Final remarks: Note that we don't actually need to "remove" SCC, we simply assume that anything that is not zero in visited is no longer part of the graph.

Example

$G =$



$G^R =$



(1) DFS in G^R and sort by post number

G, I, J, L, K, H, D, C, F, B, E, A.

(2) Run Explore from vertex G on graph G. Orange SCC is discovered.

G, I, J, L, K, H, D, C, F, B, E, A.

(3) Run Explore from D. Blue SCC is discovered.

G, I, J, L, K, H, D, C, F, B, E, A.

(4) Run Explore from C. Green SCC discovered.

G, I, J, L, K, H, D, C, F, B, E, A.

(5) Run Explore from B. Red SCC discovered.

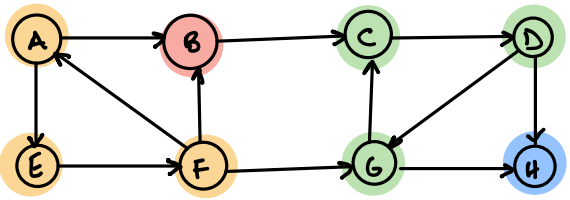
G, I, J, L, K, H, D, C, F, B, E, A.

(6) Run Explore from A. Pink SCC discovered.

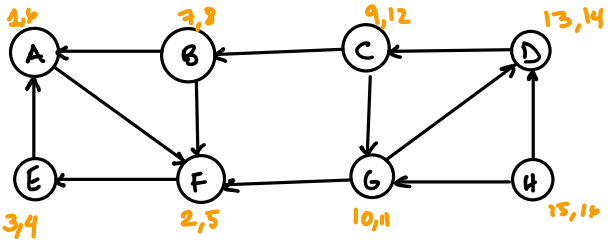
G, I, J, L, K, H, D, C, F, B, E, A.

Another SCC example

$G =$



$G_R =$



16	14	12	11	8	6	5	4
H	D	C	G	B	A	F	E

→ Sort by post number

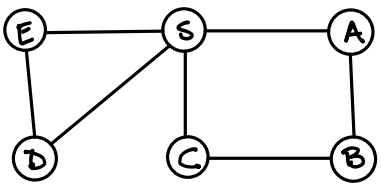
1) Run Explore from H
2) Run Explore from D
3) Run Explore from B
4) Run Explore from A

A different way of exploring a graph

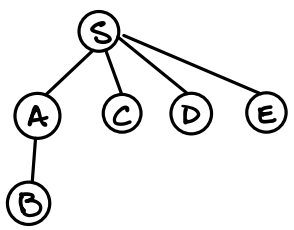
Breadth-First-Search (BFS)

Idea To find all vertices reachable from a given vertex s, we visit s first, then all vertices at distance 1, then all vertices at distance 2, etc. That is, we explore the graph layer-by-layer, with each layer corresponding to all the vertices at a fixed distance from s

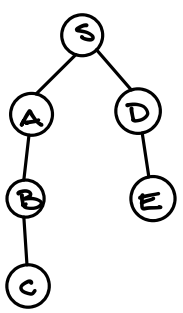
Example



BFS tree



DFS tree



⇒ Very different search patterns!

BFS - Explore

Input Graph $G=(V,E)$ (can be directed or undirected, in adjacency list or adjacency matrix)

A vertex s

A integer `color` for marking the vertices

Output All vertices reachable from s

```
visited = new int [|V|]

Q is a queue

Add s to Q

while Q ≠ empty
    v = eject(Q)
    for all edges (v, w) ∈ E
        if visited[w] == 0
            Q.add(w).
            visited[w] = color
```

Correctness Analogous (almost the same) as the explore for DFS.
Try it as an exercise!

Running Time Every vertex is placed in the queue exactly once.

The rest of the work is done by the for loop, where every edge is considered once (in the directed case) or twice (in the undirected case). [Assuming the graph is given as adjacency list]

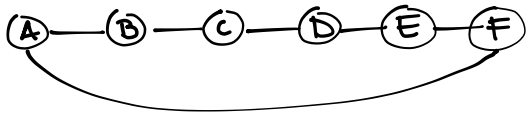
So: $O(|V|+|E|)$ like DFS.

If the graph is given in matrix form, then the for loop always take $O(|V|)$, so the overall running time is $O(|V|^2)$.

Notes

- For worst cases instances, there is no much difference btw DFS & BFS, but DFS can be more memory efficient in some cases.
- The difference is on the applications of these algorithms.
- We already saw many applications for DFS, but this algorithm is quite bad for finding shortest paths between vertices.
- BFS is quite good for this.

Example:



To go from A to F, DFS may go the long way A-B-C-D-E-F but BFS will from A to F.